IFT2245 - Systèmes d'exploitation
Travail Pratique 2 (20%)

David Quiroz Marin
DIRO - Université de Montréal

À faire en équipe de deux.
Disponible : 25/02/2015 - Remise : 25/03/2015 avant le cours

1 Introduction

Pour ce TP, vous devez implémenter en C++ *l'algorithme du banquier*, un algorithme qui permet de gérer l'allocation des différents types des ressources, tout en évitant les interblocages (*deadlocks*). Vous pouvez faire le travail demandé en équipe de deux. Le code fourni implémente un modèle de client-serveur qui utilise les prises (*sockets*) comme moyen de communication.

D'un côté l'application serveur est analogue à un système d'exploitation qui gère l'allocation des ressources, comme par exemple la mémoire, le disque dur ou les imprimantes. Ce serveur reçoit simultanément plusieurs demandes de ressources des différents clients à travers des connexions. Pour chaque connexion/requête, le serveur doit décider si les ressources peuvent être allouées ou non au client, de façon à éviter les interblocages en suivant l'algorithme du banquier.

De l'autre côte, l'application client simule l'activité de plusieurs clients dans différents fils d'exécution (*threads*). Ces clients peuvent demander des ressources si elles sont disponibles ou libérer des ressource qu'ils détiennent à ce moment.

Ce TP vous permettra de mettre en pratique quatre sujets du course différents :

1. Fils d'exécution multiples (*multithreading*)
2. Prévention des séquencements critiques (*race conditions*)
3. Évitement d'interblocages (*deadlock avoidance*)
4. Communication entre processus via des prises (*sockets*) et des *named pipes*.

2 Mise en place

Un ensemble des fichiers de code est fourni pour vous aider à commencer ce TP. Les deux applications, client et serveur, utilisent les librairies du standard POSIX pour l'implémentation des structures dont vous allez avoir besoin, notamment les *sockets*, les *threads*, les *semaphores*, les *mutex*, et les *pipes*. Ces librairies sont par defaut installées dans votre machine virtuelle.

Pour faciliter le débogage (et l'évaluation) on utilisera un fichier de configuration (`initValues.cfg`) auquel les deux applications vont avoir accès pour récupérer les paramètres de l,algorithme du banquier et des configurations additionnelles. La lecture de ce fichier est déjà codée et vous n'avez

rien à y modifier. Par contre il est nécessaire d'installer, dans votre machine virtuelle, la librairie `'libconfig'` pour le faire compiler. Il suffit d'aller dans : `System/Administration/Software Center`, chercher '`libconfig++8-dev`', et l'installer.

Tous les entêtes de fichier que vous pourriez utiliser sont déjà inclus dans le fichier '`common.h`', mais vous pouvez certainement utiliser d'autres librairies si nécessaire.

Pour commencer votre TP, le premier pas est d'installer la librairie `'libconfig'`, mettre tous les fichiers du code source fourni dans un même répertoire, et compiler chaque application avec un nom d'exécutable différent. Je vous suggère d'utiliser un éditeur de texte avancé [1] pour éditer votre code et avoir en tout temps deux consoles ouvertes pour compiler et exécuter chaque application séparément. Pour la compilation en console de chaque application, vous pouvez utiliser la commande suivante : '`g++ -lconfig++ -pthread serverThreads.cpp server.cpp -o server`' pour le serveur et : '`g++ -lconfig++ -pthread clientThread.cpp client.cpp -o client`' pour le client. Ensuite faites rouler en premier le serveur en faisant `./server` et après dans une autre console le client avec `./client`.

3 Description du projet

Dans le code source fourni, vous trouverez trois fichiers de code pour chaque application, en plus des fichiers `'common.h'` et `'initValues.cfg'` dont on a déjà parlé. Chaque application a son fichier principal `'server.cpp'` et `'client.cpp'`, ainsi que deux fichiers declasse pour définir et implémenter les fonctionnalités de chacune. La plus grande partie de votre travail se concentre dans les fichiers `'serverThreads.cpp'` et `'clientThread.cpp'` sur les fonctions délimitées par `TP2_TO_DO` et `TP2_END_TO_DO`. Ça veut pas dire que vous ne devez rien modifier ailleurs, mais c'est plutôt un guide pour commencer.

Les prises (*sockets*) qu'on utilise dans le protocole TCP/IP sont supposées de faire communiquer des applications dans différents ordinateurs sur un réseau, en utilisant la direction IP du serveur, mais pour faciliter votre implémentation on utilise la direction d'IP locale ('`localhost`'), pour se connecter sur le même ordinateur et un port arbitraire disponible (normalement plus grand que 2000) qui est spécifié dans le fichier de configuration dans `portNumber`.

Le nombre de ressources différentes disponibles dans le serveur est défini dans le fichier de configuration dans `numResources`, cette valeur aide le serveur et le client à allouer la mémoire nécessaire pour les structures des données, vous n'avez pas à vous inquiéter pour l'allocation et la libération de cette mémoire.

3.1 Application serveur (`server.cpp`)

La classe `ServerThreads` (fichiers : `serverThreads.cpp` et `serverThreads.h`) implémente le serveur, et elle inclut déjà les structures des données de base pour implémenter l'algorithme du banquier, dont on a parlé au cours de la démo.

Le serveur fourni utilise plusieurs *threads* pour recevoir les requêtes. Le nombre de *threads* ainsi que la taille de la file d'attente de chaque *thread* sont spécifiés dans le fichier de configuration avec les champs : `serverThreads` et `serverBacklogSize`. Essayez votre application avec différentes configurations de ces valeurs pour modifier la charge que votre serveur peut recevoir.

1. Des éditeurs de texte avancés permettent de mettre en évidence la syntaxe du C++, pour coder plus facilement. Vous trouverez dans le *Software Center* des exemples de tels éditeurs comme : *Geany* ou *QtCreator*

D'ailleurs, pour éviter que votre serveur roule pour un temps infini (à cause des *deadlocks*, ou d'autres erreurs), il a un temps défini comme maximal pour rouler. Ce temps, défini comme `'maxWaitTime'` est le nombre maximum de secondes que le serveur va rouler, c'est-à-dire que si le serveur n'arrive pas à finir normalement avant ce temps, l'exécution va s'arrêter automatiquement, peu importe l'état actuel. Changez cette valeur dans le fichier de configuration pour vous adapter à vos tests.

3.1.1 `ServerThreads`

La classe `ServerThreads` est la classe principale à compléter pour le serveur. Vous trouverez à l'intérieur les structures nécessaires pour gérer les *threads* et les *sockets*. Vous trouverez ainsi les structures de données que vous avez à remplir et à modifier pour implémenter l'algorithme du banquier. Le code donne des commentaires pertinents pour expliquer chaque fonction.

Quand un client fait une requête au serveur, la fonction `processRequest()` est appelée. Les clients devraient faire un certain nombre (spécifié par `numRequestPerClient`) de demandes et de libérations de ressources de façon pseudo-aléatoire (d'ailleurs, ils ne devront jamais libérer des ressources qu'ils n'ont pas). À la fin, ils doivent toujours libérer toutes les ressources qu'ils ont prises. C'est-à-dire, si par exemple les clients doivent faire N requêtes, la première (0) est toujours une demande de ressource, et la dernière ($N-1$) est toujours une libération de toutes les ressources prises par ce client. Entre ces deux, les autres peuvent être des libérations ou des demandes. Le serveur doit accepter la requête seulement si elle est valide et si elle laisse le système dans un état sécuritaire. Il peut y avoir trois types des requêtes :

- `Accepted` : Les requêtes acceptées doivent retourner au client la valeur `0`, pour lui confirmer que les ressources demandées ont été allouées. Après vous devez coder les instructions nécessaires pour mettre à jour les valeurs des ressources dans le système.
- `OnWait` : Les requêtes qui sont valides, mais qui laissent le système dans un état non-sécuritaire, ou dans le cas de manque de ressources, doivent retourner une valeur plus grande que zéro pour indiquer au client qu'il doit attendre avant d'essayer encore la même requête. Cette valeur correspond au temps d'attente, en millisecondes.
- `Invalid` : Les requêtes invalides (par example, qui dépassent le maximum permis pour ce client) doivent être refusées par le serveur en retournant au client la valeur `-1`.

Pendant l'exécution de l'application serveur et aussi de l'application client vous devez compter chacune des types des requêtes reçues et envoyées, ainsi que le nombre des clients correctement traités, utilisez les variables `countAccepted`, `countOnWait`, `countInvalid`, `countClientsDispatched` à cet effet. On peut considérer un client comme "correctement traité" après la libération des toutes ses ressources. Assurez-vous donc de ne pas faire des libérations totales avant la dernière requête de chaque client. En d'autres mots, les clients ne devront jamais rester sans ressources sinon juste avant la fin de leur exécution. Notez qu'à la fin d'exécution, ces valeurs doivent être les mêmes pour les deux applications.

Faites attention au fait que plusieurs données sont modifiées par différents clients (*threads*) "en même temps". Vous devez donc également implémenter le contrôle d'accès à ces données avec des verrous *mutex* ou *semaphores* pour éviter la bataille des ressources.

Aussi, vu que les connexions sont asynchrones, c'est fort probable que le client finisse avant le serveur, mais il ne peut certainement pas finir son exécution avant que toutes les requêtes soient traitées par le serveur. Pour le moment un `sleep(4)` est mis dans les fonctions `signalFinishToClient` et

`waitUntilServerFinishes`, mais ce comportement doit être changé.

3.2 Application client (`client.cpp`)

L'application client doit vous permettre de tester votre application serveur avec des plusieurs fils d'exécution (*threads*) qui font des requêtes de façon pseudo-aléatoire. Vous avez aussi à compléter la fonctionnalité de base de ces clients, en gardant le format des message (décrit ensuite) qui sont passés entre le serveur et le client. Vous serez évalués avec une application client qui fait plusieurs requêtes spécifiques (qui pourraient être même invalides) et votre application serveur doit répondre également à ces requêtes.

Cette application lit automatiquement du fichier configuration les paramètres suivants : le nombre de clients à simuler (`numClients`), le nombre de requêtes par client à créer et à lancer au serveur (`numRequestPerClient`) et le nombre de différentes ressources disponibles dans le serveur (`numResources`). Les valeurs que chaque client pourrait demander comme maximum de chaque ressource (la matrice `'Max'`) sont définies dans le serveur et elles sont automatiquement écrites dans un fichier que l'application client lit avant de lancer tous les *threads* des clients.

Il est important de noter que les valeurs maximales de chaque client spécifiées dans le fichier de configuration est là juste pour mieux tester des petits exemples. Vous avez à coder une façon de définir ces valeurs dans le serveur, en fonction des ressources disponibles (aussi arbitrairement définies dans le serveur), pour les cas ou le nombre de clients ou le nombre de requêtes est trop grand.

3.2.1 `ClientThread`

La classe `ClientThread` doit implémenter dans les fonctions `clientThreadCode()` et `sendRequest()` le comportement de chaque client. L'indice (commençant à 0) de chaque client est une variable privée dans la même classe. Par défaut, l'application client crée `numClients` *threads* ou clients et les exécute avec la fonction `clientThreadCode()`. Après, chaque client doit faire un ensemble de `numRequests` requêtes aléatoires tel que spécifié dans la section précédente. Les requêtes ont aussi un indice passé comme paramètre à la fonction `sendRequest()`.

Les messages envoyés par le client devront être des chaînes de caractères qui contiennent `numResources`+1 nombres entiers (`int`), séparés par des espaces. Le premier entier indique l'indice du client et les entiers suivants indiquent le nombre d'instances de chaque type de ressources demandées ou libérées. Une valeur négative indique que cette quantité de ressources est demandée, et une valeur positive indique que cette quantité de ressources est libérée. Par contre une requête ne peut jamais contenir des valeurs négatives et positives en même temps.

Par exemple, le message "`2 -5 -1 0`" représente une requête du client 2 pour demander 5 instances de la première ressource et 1 instance de la deuxième, tandis que le message "`0 0 1 2`" représente une requête du client 0, pour libérer 1 instance de la deuxième ressource et 2 de la troisième.

4 Évaluation

Votre code sera évalué de trois façons différentes : votre serveur avec votre client, votre serveur avec un client standard et votre client avec un serveur standard. Dans tous les cas l'exécution doit se faire correctement sans problèmes pour différentes configurations. Au moins un gros test va se

faire pour les trois formes d'évaluation. On considère comme gros, celui avec un grand nombre des requêtes totales (`numRequestsPerClient*numClients`) au maximum de 5000, et une grosse quantité des ressources à suivre (`numResources*numClients`) au maximum de 1000.

5 Travail à faire

5.1 Code à compléter 5%

Vous devez compléter la classe `ServerThreads` et la classe `ClientThread` (voir les commentaires du code pour plus de détails). Pour bien réussir cette partie, assurez vous que votre code compile, et vos applications échangent des messages entre elles ; que les messages, ainsi que les réponses, suivent la syntaxe demandée, et que toutes les requêtes selon le fichier de configuration se font des deux côtés.

5.2 Fonctionnement correct 10%

En plus, vous devez vérifier que les requêtes de l'application client ne mettent pas le système en *deadlock*, et vous devez garantir que l'intégrité des ressources dans le système est préservée. À la fin de l'exécution de l'application client, le serveur doit revenir à son état initial. Finalement, peu importe les configurations utilisées, les deux applications doivent finir correctement leur exécution.

5.3 Rapport (deux pages maximum) 5%

Décrivez brièvement le fonctionnement de votre solution, et ensuite répondez aux questions suivantes. N'oubliez pas d'écrire les noms des deux membres de l'équipe.

1. Comment vous pouvez garantir que votre système n'arrivera jamais à être en *deadlock* ?
2. Donnez un exemple d'une situation où les données pourraient être corrompues. Comment vous avez réglé ce problème ?
3. Quelle solution vous avez implémentée pour bien synchroniser la fin d'exécution des deux programmes ?, pourquoi c'est important ? et pourquoi un simple `'sleep()'` ne suffit pas dans tous les cas ?

6 Format des documents à remettre

Mettez votre rapport en format `pdf` et tous les fichiers de code que vous avez modifiés dans une archive **PRENOM_NOM_TP2.zip** (ou .tar.gz) et déposez-la dans StudiUM (soumettre seulement une archive par équipe).

7 Liens utiles (en anglais)

- Tutoriel sur *sockets* - `http://www.linuxhowtos.org/C_C++/socket.htm`
- Documentation sur les *threads* et *mutex* du standard POSIX - `https://computing.llnl.gov/tutorials/pthreads/#Mutexes`
- Pipes examples - `http://tuxthink.blogspot.ca/2012/02/inter-process-communication-using-named html`